UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/893,202	06/27/2001	Bernard Baldwin	6311-022	3859

21890 7590 01/05/2005

PROSKAUER ROSE LLP
PATENT DEPARTMENT
1585 BROADWAY
NEW YORK, NY 10036-8299

EXAMINER	
JAROENCHONWANIT, BUNJOB	

ART UNIT	PAPER NUMBER
2143	

DATE MAILED: 01/05/2005

Please find below and/or attached an Office communication concerning this application or proceeding.

Offic Action Summary	Applicati n N .	Applicant(s)
	09/893,202	BALDWIN ET AL.
	Examiner	Art Unit
	Bunjob Jaroenchonwanit	2143

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If the period for reply specified above is less than thirty (30) days, a reply within the statutory minimum of thirty (30) days will be considered timely.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on 27 September 2001.

2a)☐ This action is **FINAL**. 2b)☒ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) 1-32 is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) 1-3,6,9,14-16,19,22,27 and 30 is/are rejected.

7)☐ Claim(s) 4-5, 7-8, 10-13, 17-18, 20-21, 23-26, 28-29, 31-32 is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☐ The specification is objected to by the Examiner.

10)☐ The drawing(s) filed on _____ is/are: a)☐ accepted or b)☐ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☐ All b)☐ Some * c)☐ None of:

1.☐ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____.

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

1) ☒ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☒ Information Disclosure Statement(s) (PTO-1449 or PTO/SB/08) Paper No(s)/Mail Date 12/17/01.
4) ☐ Interview Summary (PTO-413) Paper No(s)/Mail Date. _____.
5) ☐ Notice of Informal Patent Application (PTO-152)
6) ☐ Other: _____.

DETAILED ACTION

Claim Rejections - 35 USC § 102

1. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (e) the invention was described in (1) an application for patent, published under section 122(b), by another filed in the United States before the invention by the applicant for patent or (2) a patent granted on an application for patent by another filed in the United States before the invention by the applicant for patent, except that an international application filed under the treaty defined in section 351(a) shall have the effects for purposes of this subsection of an application filed in the United States only if the international application designated the United States and was published under Article 21(2) of such treaty in the English language.

2. Claims 1-3, 6, 9, 14-16, 19, 22, 27 and 30 are rejected under 35 U.S.C. 102(e) as being Anticipated by Gurevich (US. 6,499,036).

3. Regarding claims 1-2, 6, 9, 14-16, 19, 22, 27 and 30, Gurevich discloses a computer system including software and, implementing method, in which capable of providing distributed functionality to a plurality of clients comprising: a first provider server having a function provider module therein (Fig. 2, 210); a data store connected to the function provider module and containing function information defining at least one function object, each function object associated with a function and comprising a function name element and specifying at least one parameter (DOM 115); the function provider module being configured to: in response to receipt of a first type request from a requester, return a set of function objects to the requester (Fig. 9, 662, illustrated list of functional objects in response to first type request); in response to receipt of a second type request from the requester containing a function object having defined parameter values stored there, evaluate the function associated with the function object (Fig. 9, 910), modify the received function object to include results of the function evaluation, and return

the modified functi n object to the requestor (return result 900). Regarding claims 2, the system of claim 1, wherein each function object further comprises a function ID and an identity of a respective provider server (see example Fig. 10-1020; function GetBankCard comprises script which include ID and identity, "BIN", $ID).

4. Regarding claims 3, Implicitly, Gurevich discloses the data can be returned from any appropriated data source (130). Although Gurevich does not explicitly state a third server or so forth, but it would have been obvious to one of ordinary skill in the art at the time of the invention was made that Gurevich inventive concept can be replicated to any number servers or service providers as desirable.

5. Claims 4-5, 7-8, 10-13, 17-18, 20-21, 23-26, 28-29 and 31-32 are objected to as being dependent upon a rejected base claim, but would be allowable if rewritten in independent form including all of the limitations of the base claim and any intervening claims.

6. Any inquiry concerning this communication or earlier communications from the examiner should be directed to Bunjob Jaroenchonwanit whose telephone number is (571) 272-3913. The examiner can normally be reached on 8:00-17:00.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, David Wiley can be reached on (571) 272-3923. The fax phone number for the organization where this application or proceeding is assigned is 703-872-9306.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status informati n for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).

Bunjob Jaroenchonwanit
Primary Examiner
Art Unit 2143

/bj
11/20/04

Notice of References Cited

Application/Control No.	Applicant(s)/Patent Under Reexamination	
09/893,202	BALDWIN ET AL.	
Examiner	Art Unit	Page 1 of 1
Bunjob Jaroenchonwanit	2143	

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
	A	US-6,499,036 B1	12-2002	Gurevich, Michael N.	707/103R
	B	US-6,345,292 B1	02-2002	Daugherty et al.	709/214
	C	US-5,446,896 A	08-1995	Hegarty et al.	707/1
	D	US-6,067,548 A	05-2000	Cheng, Edward C.	707/103R
	E	US-5,692,157	11-1997	Williams, Antony S.	709/246
	F	US-5,913,064 A	06-1999	Chen, Chi-Chung K.	717/108
	G	US-5,859,972 A	01-1999	Subramaniam et al.	709/203
	H	US-5,371,848 A	12-1994	Casey et al.	715/823
	I	US-5,255,358 A	10-1993	Busboom et al.	715/843
	J	US-6,288,726 B1	09-2001	Ballard, Dean D.	345/468
	K	US-6,526,129 B1	02-2003	Beaton et al.	379/90.01
	L	US-6,556,963 B1	04-2003	Tetzlaff, Linda S.	704/9
	M	US-6,769,019 B2	07-2004	Ferguson, Xavier	709/219

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	Carr et al., "Compiling Distr8buted C++", IEEE, 1993, pages 496-503.
	V	
	W	
	X	

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

OIPE JC106
DEC 1 7 2001
PATENT & TRADEMARK OFFICE

FORM PTO-1449 U.S. Department of Commerce Patent and Trademark Office	Serial No. 09/893,202
INFORMATION DISCLOSURE STATEMENT	Inventors: Baldwin et al.
(Use several sheets if necessary)	Docket No. 6311-022

U.S. PATENT DOCUMENTS

EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUBCLASS	FILING DATE IF APPROPRIATE
BYE		5,471,596	11/28/95	Brown, III	395	375	11/10/94
		5,881,380	3/9/99	Mochizuki et al.	707	102	4/11/96

FOREIGN PATENT DOCUMENTS

EXAMINER INITIAL	DOCUMENT NUMBER	DATE	COUNTRY	TRANSLATION

RECEIVED
DEC 1 8 2001
Technology Center 2100

OTHER DOCUMENTS

EXAMINER INITIAL		AUTHOR, TITLE, DATE, PERTINENT PAGES, ETC.
BY		International Search Report dated October 5, 2001 for PCT/US01/20526

EXAMINER B. [signature]	DATE CONSIDERED 11/20/04

EXAMINER: Initial if citation considered, whether or not citation is in conformance with M.P.E.P. 609; draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

Form PTO-1449 [6-4]

NYA 506591.1

Compiling Distributed C++

Harold Carr, Robert R. Kessler, Mark Swanson
Department of Computer Science
University of Utah
Salt Lake City, Utah, 84112

Abstract

Distributed C++ (DC++) is a language for writing parallel applications on loosely coupled distributed systems in C++. Its key idea is to extend the C++ class into 3 categories: gateway classes which act as communication and synchronization entry points between abstract processors, classes whose instances may be passed by value between abstract processors via gateways, and vanilla C++ classes. DC++ code is compiled to C++ code with calls to the DC++ runtime system. The DC++ compiler wraps gateway classes with handle classes so that remote procedure calls are transparent. It adds static variables to value classes and produces code which is used to marshal and unmarshal arguments when these value classes are used in remote procedure calls. Value classes are deep copied and preserve structure sharing. This paper shows DC++ compilation and performance.

1 Introduction

DC++ is designed to exploit loosely coupled distributed systems built by interconnecting multiple workstations through a local area network. DC++ is a distributed version of C++ [8] (this paper assumes knowledge of C++). DC++ provides a small number of simple extensions to C++: 2 built-in classes: `DcDomain` and `DcThread`; and 2 new categories of class instance usage: *gateways* between domains, and "value" instances which may be passed between domains through gateway member function invocation and return. The DC++ language is discussed in [6, 5]. This paper shows how DC++ is compiled. We will use the bounded buffer problem [1] as a running example throughout this paper. We include performance measurements for this example.

2 Domains — Abstract Processors

DC++ supports parallelism by providing 2 types: *domains* and *threads* ([13, 17]). A domain is a logically encapsulated address and control space, an *abstract processor*. A domain is similar to a monitor [11]: they ensure *mutual exclusion synchronization* by enforcing the rule that only one thread of control may be active in a domain at a time. If another thread attempts entry to an occupied domain, that thread will be queued on a FIFO queue for later entry when the domain becomes vacant. It differs from a monitor in that domains may be dynamically created and deleted. Further, a domain by itself does not have any entry points. Entry points may be dynamically created and deleted by creating *gateway* class instances into domains. A domain is created by specifying a physical processor number (0-based) on which to allocate the domain. For the bounded buffer example we will create 6 domains: one each for the 3 producers, one for the buffer, and one each for 2 consumers:

```
const num_producers = 3;
const num_consumers = 2;
DcDomain* pd[num_producers];
DcDomain* cd[num_consumers];
DcDomain* bd = new DcDomain(num_producers +
                            num_consumers);
for (int i = 0; i < num_producers; ++i)
    pd[i] = new DcDomain(i);
for (i = 0; i < num_consumers; ++i)
    cd[i] = new DcDomain(i + num_producers);
```

Since there may be more domains than actual physical processors, the domain is allocated on `i % DcNodeCount()`, where `DcNodeCount()` returns the 1-based number of physical processors available to the specific execution. This means that more than one domain may be explicitly (by the programmer) or implicitly (by the domain allocator) created on a processor. The DC++ runtime system supports multitasking so the programmer need not be concerned with these details.

1063-6374/93 $03.00

3 Gateways — Domain Entry P ints

A gateway is a system-wide unique "pointer" to an object created in a specific domain. It is treated as an ordinary C++ object: member function invocations on gateway objects result in remote procedure calls (RPCs) if that gateway resides in a different domain than the domain from which it is invoked. Otherwise a vanilla C++ member function invocation results. Since RPCs look identical to vanilla C++ member function invocations, redistribution of gateways is possible without program modification (modulo synchronization characteristics of the algorithm).

A gateway class is declared like a vanilla C++ class, except it inherits from DcGateway:

```
class Producer : public DcGateway {
  int     num_items; // Number items to produce.
  Buffer* buffer;    // Buffer to put them in.
public:
  Producer(int i, Buffer* b) {
    num_items = i, buffer = b;
  }
  void Produce() {
    while (num_items--) {
      buffer->Deposit(new Derived(num_items));
    }
  }
};
class Buffer : public DcGateway {
  unsigned max_items; // Capacity of buffer.
  Derived** slots; // Array to contain items.
  unsigned head;  // Where producer puts items.
  unsigned tail;  // Where consumer gets items.
  unsigned size;  // Number of items in buffer.
public:
  Buffer(int i){
    head = tail = size = 0;
    max_items = i;
    slots     = new Derived*[max_items];
    MakeDelayQueue(Buffer::Deposit);
    DQOpen(Buffer::Deposit);
    MakeDelayQueue(Buffer::Fetch);
  }
  void Deposit(Derived* i){
    slots[tail] = i;
    size++;
    tail = (tail + 1) % max_items;
    if (size > 0)
      DQOpen(Buffer::Fetch);
  }
  Derived* Fetch(){
    Derived* retval = slots[head];
    size--;
    head = (head + 1) % max_items;
    if (size < max_items)
      DQOpen(Buffer::Deposit);
    return retval;
  }
};
class Consumer : public DcGateway {
  int     num_items; // Number items to consume.
  Buffer* buffer;    // Buffer to get them from.
public:
  Consumer(int i, Buffer* b) {
    num_items = i, buffer = b;
  }
  void Consume() {
    while (num_items--) {
      Derived* i = buffer->Fetch();
      if (i->Data() == 0)
        return;
      spin();
    }
  }
};
```

The buffer has explicit *delay queues* associated with its Deposit and Fetch methods. Delay queues provide *condition synchronization*. If the delay queue is open, calls to the method proceed as normal. If the delay queue is closed then the calls are queued on a FIFO queue for later entry to the method when the queue is opened. A call (thread) waiting on a method's delay queue is not considered to have entered the domain. They are used in this example to ensure that producers only deposit items when there is room in the buffer, and that consumers only fetch items when there is one or more available.

A gateway instance is created in a specific domain by providing an optional domain argument as the first argument to the gateway's constructor. If not present the current domain is used:

```
Buffer*   b  = new Buffer(bd, 8);
Producer* p[num_producers];
Consumer* c[num_consumers];
for (i = 0; i < num_producers; ++i)
    p[i] = new Producer(pd[i], 25, b);
for (i = 0; i < num_consumers; ++i)
    c[i] = new Consumer(cd[i], 25, b);
```

Note that the type passed between the producer, buffer and consumer is Derived*. This type is a user defined "value" class and will be discussed later. Also note that the optional domain argument is not declared in the user's gateway class definition. This is handled by the compiler.

4 Threads

Concurrency is achieved by creating multiple threads of control. In the bounded buffer example,

threads are created for the producers and consumers, whereas the buffer is passively enclosed in a domain to ensure mutually exclusive access to it:

```
for (i = 0; i < num_consumers; ++i)
    new DcThread(c[i]->Consume());
for (i = 0; i < num_producers; ++i)
    new DcThread(p[i]->Produce());
```

Threads may return values when they terminate. These values may be used by the thread which created the new thread, and/or the termination of the created thread may be detected by the creating thread. This feature is not used in this example.

5 Compiling Gateways

The fundamental idea is to transform the program so that all references to user defined gateway classes are changed to references to compiler generated *handle classes*. Each user gateway class has an associated handle class which intercepts all method invocations. The handle class determines if the gateway method being invoked is for an instance in the same domain as the invoker. If so, it does a normal C++ method invocation. Otherwise, it marshals the arguments and does an RPC to the domain in which the gateway resides. Upon return it unmarshals the return value. The compiler generated handle class frees the programmer from these details and allows gateways to be redistributed without program modification. This section shows the details of the gateway compilation process.

A unique tag is created for all gateway classes in a program:

```
enum CLASS_TAGS {
        Consumer_TAG,
        Buffer_TAG,
        Producer_TAG,
};
```

These tags are used to indicate the type of object being made when that object is created remotely.

Each user gateway class is wrapped by a handle class:

```
class Buffer_W : public gateway {
public:
  Buffer_W(int i)
    : gateway(new Buffer(i)) {}
  Buffer_W(DcDomain* d, int i)
    : gateway(d,
            MakeRemoteObject(Buffer_TAG,
                                  d, i)) {}
  Derived*  Fetch();
  void      Deposit(Derived*  );
};
```

For each constructor in the user class, 2 constructors are created in the handle class: one with a type signature identical to the user's definition, and one which adds a domain argument as the first parameter. In this way gateways may be created in the same domain or between domains.

The handle class inherits from gateway, whose definition is:

```
class gateway {
private:
  DcDomain* domain;  // domain where object lives
  DcObject* remote;  // remote object
  void*     local;   // local object
protected:
  DcDomain* domainGid() { return domain; }
  DcObject* remoteObj() { return remote; }
  void*     localObj()  { return local; }
  void*     localP()    { return local; }
  gateway(void* v) :
    domain(NULL), remote(NULL), local(v)    {}
  gateway(DcDomain* did, DcObject* oid) :
    domain(did),  remote(oid),  local(NULL) {}
};
```

When creating objects derived from gateway, a gateway is returned which either points to an instance in the domain or to an instance in a remote domain.

When the handle class constructor is given an optional domain it constructs the actual object on a remote node via the compiler generated MakeRemoteObject routine:

```
DcObject* MakeRemoteObject(unsigned type){
    void* v = NULL;
    switch (type) {
    case Consumer_TAG :
        v = new Consumer; break;
    case Buffer_TAG   :
        v = new Buffer;   break;
    case Producer_TAG :
        v = new Producer; break;
    default:
        DcError("Unknown object type");
    break;
    }
    return RegisterObject(v, DcThisDomain());
}
```

RegisterObject is a DC++ runtime system routine which places the actual pointer to the newly created object into an OutTable table, a table of entities which may be used remotely. It returns a DcObject* which is a special pointer type which is unique and valid

between domains. The bits of this pointer indicate the node on which the object resides and the index of the actual object pointer in that node's `OutTable` table.

This example is incomplete in that it doesn't show how arguments to constructors are handled remotely, and it only shows one constructor per class. If there is more than one constructor they are distinguished by their type signature. Constructor arguments are handled in a manner similar to arguments to methods (shown below).

A tag is created for each public method in the handle class (constructor tags not shown in example):

```
enum BufferPTR_METHOD_TAGS {
        DerivedptrBufferFetch_TAG,
        voidBufferDepositDerivedptr_TAG,
};
```

For each method in the user's original gateway definition an associated method is defined in the handle class:

```
void Buffer_W::Deposit(Derived* a3){
  if (localP())
    return ((Buffer_W*)localObj())->Deposit(a3);
  MsgBufId f = MakeMsgBuf(100);
  SetMsgBuf(f, 0, remoteObj());
  SetMsgBuf(f, 1, voidBufferDepositDerivedptr_TAG);
  PACK_voidBufferDepositDerivedptr_PARMS(f, &a3);
  ApplyWithinDomain(REMOTE_Buffer_HANDLER,
                    f, domainGid());
  DeleteMsgBuf(f);
}
Derived* Buffer_W::Fetch(){
  if (localP())
    return ((Buffer_W*)localObj())->Fetch();
  MsgBufId f = MakeMsgBuf(2);
  SetMsgBuf(f, 0, remoteObj());
  SetMsgBuf(f, 1, DerivedptrBufferFetch_TAG);
  MsgBufId fr = (MsgBufId)
        ApplyWithinDomain(REMOTE_Buffer_HANDLER,
                          f, domainGid());
  DeleteMsgBuf(f);
  Derived* result;
  UNPACK_DerivedptrBufferFetch_RETVAL(fr, &result);
  DeleteMsgBuf(fr);
  return result;
}
```

These handle methods are how transparent RPCs are achieved. If a `Buffer` handle instance is created in the same domain as the invoker of its constructor then a pointer to that local object is installed in the gateway's `local` method variable. When a handle method is invoked it first checks to see if the object is local. If so it avoids the overhead of RPC by invoking the local method. Otherwise it marshals the remote object reference, the method tag, any arguments, and then calls the RPC handler in the domain for that handle class via `ApplyWithinDomain`. `ApplyWithinDomain` is the blocking remote procedure call routine in the DC++ runtime system. When the RPC returns, it unmarshals the return value into a message buffer. The operation of compiler generated `PACK` and `UNPACK` marshaling routines is discussed later. They use runtime message buffers (`MsgBufId`) to contain object references, method tags, argument values, and return values.

A remote method handler is created for each handle class. The method tags are used to dispatch to the appropriate method when handling RPCs:

```
void* REMOTE_Buffer_HANDLER(MsgBufId f){
  Buffer* r = (Buffer*) OutTable(MsgBuf(f, 0));
  BufferPTR_METHOD_TAGS m = MsgBuf(f, 1);
  switch (m) {
  case DerivedptrBufferFetch_TAG : {
    DeleteMsgBuf(f);
    Derived* a4 = r->Fetch();
    MsgBufId fr =
          PACK_DerivedptrBufferFetch_RETVAL(&a4);
    return (void*) fr;
  }
  case voidBufferDepositDerivedptr_TAG : {
    DeleteMsgBuf(f);
    Derived* a3;
    UNPACK_voidBufferDepositDerivedptr_PARMS(f,
                                             &a3);
    r->Deposit(a3);
    return NULL;
  }
  default:
    DcError("Unknown method");
  }
}
```

This routine dispatches to the appropriate method call, marshals the arguments from the message buffer and calls the associated user operation. Return values are placed in a message buffer to be used on the other end of the RPC.

6 Value Objects

Systems such as [2] provide primitives for sending and receiving data between processes, but require the programmer to pack and unpack aggregate data. Besides the gateway classes, which marshal and unmarshal arguments and return values, DC++ provides "value" classes so that programs may pass deep,

structure-preserving copies of value class instances between domains. A value object is a class which inherits from the built-in DC++ class `DcValue`:

```
class Base : public DcValue {
        int data;
public:
        Base(int i = 0) : data(i) { }
        int Data() { return data; }
};
class Contained : public DcValue {
        char c;
public:
        Contained(char _c = 't') : c(_c) { }
};
class Derived : public Base {
        Contained mi, *mp1, *mp2;
        double    d;
public:
        Derived(int    _i = 0,
                double _d = 0.0,
                char   c1 = 'w',
                char   c2 = 'x')
                : Base(_i),
                  mi(c1),
                  d(_d),
                  mp1(new Contained(c2)),
                  mp2(mp1) { }
        ~Derived() { delete mp1; }
};
```

(Note that member data `mp1` and `mp2` point to the same instance.)

Inheriting from `DcValue` indicates that when one of these objects is passed as an argument to, or return value from a gateway member function, it is to be totally copied. This means that any objects, pointers to objects, or built-in data types contained within the passed object, must be recursively copied *and* any structure sharing present via pointers must be preserved. Any member data objects or member data pointers to objects, must be objects which also derive from `DcValue` so the compiler can add the necessary support for the complete copy operation. The compiler handles C++ scalar types automatically.

Value objects may be passed-by-value freely between domains via gateway member function arguments and return values. This is seen in the example by the calls to the `Buffer` methods `Deposit` and `Fetch` which accept and return instances of the user defined `Derived` class. These routines actually pass and return `Derived*`. In this case, the object is still copied and the pointer to the newly created object on the receiving end is used rather than the original pointer.

7 Compiling Value Objects

7.1 Runtime Type Information

The compiler adds static variables and virtual functions to each user class which inherits from `DcValue`:

```
class Base : public DcValue {
        DECLARE_TYPE(Base);
        int data;
 public:
        Base(int i = 0) : data(i) { }
        int Data() { return data; }
};
```

The `DECLARE_TYPE` macro:

```
#define DECLARE_TYPE(name) \
 public: \
  virtual const TypeInfo* Type() const \
          { return &info_obj; } \
  static const TypeInfo* Info() \
          { return &info_obj; } \
  virtual void    Writer(ostream&) const ; \
  static DcValue* Reader(istream&); \
  static name*    Read(istream& s) \
          { return (name*)DcValue::Read(s); } \
  name(istream&); \
private: \
  static  const TypeInfo  info_obj \
```

defines `Type` and `Info` methods used to obtain type information at runtime. This information is contained in the private static member data `info_obj`. `Type` is used to get this information given any instance which derives from `DcValue` (e.g., `some_instance->Type()`). `Info` is used to get this information if the type is known before hand (e.g., `Base::Info()`). These are used to obtain type-specific reader functions used when sending objects between domains.

7.2 Sending Objects Over Streams

The `Writer`, `Reader`, `Read` methods and the `name(istream&)` constructor are used to convert objects to/from linear byte streams for transmission and reception between domains. The compiler generates these for each type that directly or indirectly inherits from `DcValue`:

```
DEFINE_TYPE(Base);
DEFINE_TYPE(Contained);
DEFINE_TYPE(Derived);
```

The `DEFINE_TYPE` macro:

```
#define DEFINE_TYPE(name) \
  DcValue*        name::Reader(istream& s) \
```

```
        { return new name(s); } \
  const TypeInfo name::info_obj(STRINGIFY(name), \
                                 name::Reader) \
```

defines the Reader method which uses the constructor from istream to create an instance of the class given a linear stream of bytes. It also initializes the static info_obj method data to contain the name of the class and a pointer to the class reader.

The compiler generates a Writer method and an istream constructor for each value class:

```
Base::Base(istream& s) : DcValue(s) {
        s >> data;
}
void Base::Writer(ostream& s) const {
        s << data << endl;
}
Contained::Contained(istream& s) : DcValue (s) {
        s >> c;
}
void Contained::Writer(ostream& s) const {
        s << c << endl;
}
Derived::Derived(istream& s)
        : Base(s),
          mi(s),
          mp1(Contained::Read(s)),
          mp2(Contained::Read(s))
{
        s >> d;
}
void Derived::Writer(ostream& s) const {
        Base::Writer(s);
        mi.Writer(s);
        mp1->Write(s);
        mp2->Write(s);
        s << d << endl;
}
```

Derived types first call the static Writer for their base classes (only one in this example). Then they write out instances using the Writer method of the instance. Pointers to instances are written using the Write method. Write keeps a table of pointers to preserve structure sharing. Built-in data types are written and read to and from a stream with the normal C++ iostream operators. Data is written and read in identical order.

7.3 Argument Marshaling

The compiler generated Buffer_W::Deposit method receives a Derived* as an argument. It converts this to a linear byte stream via the generated PACK_voidBufferDepositDerivedptr_PARMS routine. This routine uses the write operations to write into an output string stream:

```
void PACK_voidBufferDepositDerivedptr_PARMS(
                        MsgBufId f, Derived** a3){
        ostrstream strout;
        (*a3)->Write(strout);
        PackMsgBuf(strout.str());
}
```

Write puts a linear representation of the contents of the instance of Derived into the string output stream strout. The byte stream component of the stream is obtained with the C++ standard strstream.h routine strout.str(). This is given to the message buffer for transmission to another domain (potentially over a LAN). (It is the message buffer's job to delete the byte stream when it is no longer needed.)

On the receiving end, REMOTE_Buffer_HANDLER uses UNPACK_voidBufferDepositDerivedptr_PARMS to convert this byte stream representation back into an instance in memory:

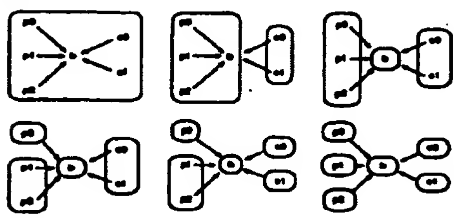
```
void UNPACK_voidBufferDepositDerivedptr_PARMS(
                        MsgBufId f, Derived** a3){
        int* fb = MsgBufBase(f);
        char* fbp = (char*) (fb + 2);
        istrstream instr(fbp, strlen(fbp) + 1);
        *a3 = Derived::Read(instr);
}
```

The message buffer's byte stream is obtained via MsgBufBase. It skips past the first two words (which contain the object reference and method tag) and uses the resulting stream and its length to create a standard strstream.h istrstream (input stream string). This stream is given to the static Derived class stream reader.

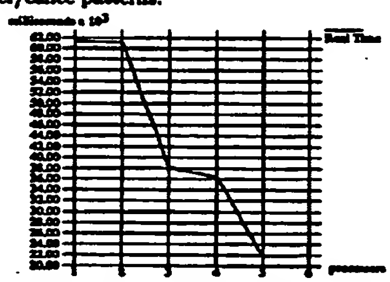
Derived::Read gives the stream to the input stream constructor for that type Derived::Derived(istream&) (defined above by DEFINE_TYPE). This in turn passes the stream first to the Base constructor and then the Contained constructor via Base(s) and mi(s) respectively. Since the next two fields, mp1 and mp2 are pointers, these are initialized from the stream via explicit calls to their static reader methods: Contained::Read. These explicit calls are necessary to detect structure sharing, whereas constructors are called when the method data contains an instance rather than a pointer. In this example, mp1 and mp2 do point to the same structure on the sending end, so this will be preserved on the receiving end. The reader routines keep a table of pointers for this purpose.

8 Performance Measurements

These measurements were taken on networked HP Series 9000, Model 370 Workstations. Communication between processors is via BSD sockets. For the measurements we created three producer threads, a single buffer, and two consumer threads. The producers produce 25 items each. The buffer capacity is eight. The consumers consume until they have consumed 25 items, or the `Data` item is 0. The consumers spin (representing some work) for a period before attempting another fetch. The allocation of domains/gateways to processors, and the interprocessor communication pattern is:



pN, b and cN represent the producer, buffer, and consumer domains and their associated gateways. The circles represent real processors. The arrows represent caller/callee patterns.



The base case for both is for the producers, the buffer and the consumers to all multitask on a single processor. Speedup is achieved at three processors once the buffer is allocated to its own processor. At that point the consumers do not have to wait for a producer multitasking with the buffer to obtain access to the buffer. At four a slight speedup is achieved, but the producers must wait for the consumers to open space in the buffer. The best speed is obtained at five processors when the consumers each have their own processor. Adding another processor slightly increases the time since the producers produce faster than consumers can consume. The increased time is due to interprocessor communication overhead.

Other allocation patterns are possible, but are not the subject of this paper. Automated resource allocation is handled by [9]. Other problems such as the dining philosophers problem programmed in DC++ have shown realistic speedup [5].

9 Other Parallel C++ Languages

Another way to provide concurrency is to define an abstract class, `Task`, that implements the task abstraction. The constructor for `Task` creates a thread to animate the task. User-defined task classes may inherit from `Task`. This approach has been used to provide coroutine facilities [16, 14] and simple parallel facilities [7, 3, 4].

This "active object" approach has problems. (1) Code to start the task body appears in `Task`'s constructor. In C++, that code is executed first, before the constructors of any derived classes. Therefore, the new thread must be created in a blocked state, and must be unblocked after derived constructors finish. (2) Placing the task's body in its constructor is ruled out by inheritance, since a derived class must still execute the private data initialization of its base class but override the base class task body. A solution is to make the body a virtual `main` member function so derived tasks can replace it. (3) During initialization, while `Task`'s constructor is executing, the new task is considered to be an instance of class `Task`, not the actual derived task class being instantiated. Therefore, calling `main` in `Task`'s constructor will not execute the correct task body. DC++ avoids these problems by using "passive" objects. In fact, DC++ unbundles everything: threads are distinct from domains, which are themselves distinct from objects and gateways.

DC++ gateways (which provide RPC) are similar to "handles" in ESP [15]. However, use of ESP handles may require casting and may be visible to the programmer, whereas type-safety and the mechanics of gateways are handled by the DC++ compiler. CC++ [12] uses a combination of "global pointers" and "logical processor objects" to provide RPC. CC++ also employs single-assignment "sync" variables and the notion of "atomic" functions, whereas DC++ uses domains for synchronization and atomicity.

Systems such as [2], require the programmer to marshal values passed to remote procedures by hand. DC++ extends C++ such that values passed between domains are automatically (un)marshalled. DC++ extends this capability to user defined types through the concept of "value" classes, which may use all in-

heritance mechanisms. DC++ value types are similar to ObjectIO in the NIH Class Library [10].

10 Conclusions and Future Work

We have designed a distributed version of C++ based on concurrency mechanisms developed in our previous work in Concurrent Scheme [18]. We are continuing development of the DC++ compiler so it will fully automate the compilation process and detect illegal DC++ usages. The DC++ runtime system runs on homogeneous workstations. We plan to extend it to handle heterogeneous systems. The current performance measurements are very encouraging.

Acknowledgements

This research was funded by Hewlett-Packard.

References

[1] G. R. Andrews and F. B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[2] A. Beguelin et al. A User's Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, 1991.

[3] B. Bershad, E. D. Lazowska, and H. M. Levy. Presto: A System for Object-Oriented Parallel Programming. *Software Practice and Experience*, 18(8), August 1988.

[4] P. A. Buhr and A. Stroobosscher. The μSystem: Providing Light-Weigh Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software Practice and Experience*, 20(9):929–963, September 1990.

[5] H. Carr. Distributed Object-Oriented Programming With C++. Center for Software Science Op Note, University of Utah, Department of Computer Science, 1993.

[6] H. Carr, R. R. Kessler, and M. Swanson. Distributed C++. *ACM SIGPLAN Notices*, 28(1):81, January 1993.

[7] T. W. Doeppner and A. J. Gebele. C++ on a Parallel Machine. In *Proceedings and Additional Papers C++ Workshop*, pages 94–107, November 1987.

[8] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[9] J. D. Evans and R. R. Kessler. Allocation of Parallel Programs With Time Variant Resource Requirements. In *Proceedings of the 1993 International Conference on Parallel Processing*. International Conference on Parallel Processing, The Pennsylvania State University Press, 1993.

[10] K. E. Gorlen, S. M. Orlow, and P. S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, 1990.

[11] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[12] C. Kesselman and K. M. Chandy. Compositional C++: Compositional Parallel Programming. Unnumbered Technical Report, California Institute of Technology, 1993.

[13] R. R. Kessler, H. Carr, L. Stoller, and M. Swanson. Implementing Concurrent Scheme for the Mayfly Distributed Parallel Processing System. *Lisp and Symbolic Computation Journal*, 5(1/2):73–93, May 1992.

[14] P. Labreche. A Real-Time Executive with Multiparty Interactions in C++. *SIGPLAN Notices*, 25(4):20–32, April 1990.

[15] W. J. Leddy, K. S. Smith, and A. Chatterjee. The Experimental Systems Software Environment for Distributed Object Execution. MCC Technical Report ACT-ESP-015-91, Microelectronics and Computer Technology Corporation, Austin, Texas, November 1990.

[16] J. E. Shopiro. Extending the C++ Task System for Real-Time Control. In *Proceedings and Additional Papers C++ Workshop*, pages 77–94, November 1987.

[17] M. Swanson. Concurrent Scheme Reference. *Lisp and Symbolic Computation Journal*, 5(1/2):95–104, May 1992.

[18] M. Swanson and R. Kessler. Concurrent scheme. In R. Halstead Jr. and T. Ito, editors, *Parallel Lisp: Languages and Systems, Lecture Notes in Computer Science Vol 441*, pages 200–234. Springer-Verlag, 1990.